

BSP Sorting: An experimental Study

Alexandros V. Gerbessiotis
CS Department
New Jersey Institute of Technology
Newark, NJ 07102.

Constantinos J. Siniolakis
The American College of Greece
6 Gravias St.
Aghia Paraskevi
Athens, 15342
Greece.

August 29, 2014

Abstract

The Bulk-Synchronous Parallel model of computation has been used for the architecture independent design and analysis of parallel algorithms whose performance is expressed not only in terms of problem size n but also in terms of parallel machine properties. In this paper the performance of implementations of deterministic and randomized BSP sorting algorithms is examined. The deterministic algorithm is the one in [22, 28] that uses deterministic regular oversampling and parallel sample sorting and is augmented to handle duplicate keys transparently with optimal asymptotic efficiency. The randomized algorithm derives from the algorithm in [21] that is sample-sort based and uses oversampling and the ideas introduced with the deterministic algorithm. The resulting randomized design, however, works differently from traditional parallel sample-sort based algorithms and is also augmented to transparently handle duplicate keys with optimal asymptotic efficiency thus eliminating the need to tag all input keys and to double communication/computation time. Both algorithms are shown to balance the work-load evenly among the processors and the use and precise tuning of oversampling that the BSP analysis allows combined with the transparent duplicate-key handling insures regular and balanced communication. Both algorithms have been implemented in ANSI C and their performance (scalability and efficiency issues) has been studied on a distributed memory machine, a Cray T3D. The experimental results obtained from these implementations are reported in this work. The validity of the theoretical model is also tested; based on the theoretical performance of each algorithm under the BSP model and the BSP parameters of a Cray T3D, it is possible to estimate the actual performance of the implementations based on the theoretical performance of the designed algorithms.

1 Introduction

One of the early attempts to model a parallel computer has been the Parallel Random Access Machine (PRAM) [19] which is one of the most widely studied abstract models of a parallel computer. The advantage of the PRAM model in terms of algorithm design is its simplicity in capturing parallelism and abstracting away the communication requirements of parallel computing such as communication latency, block transfers, memory and network conflicts during routing, bandwidth of interconnection networks, interprocessor communication, memory management and synchronization. In addition, most of the algorithms designed for the PRAM model work under unlimited parallelism assumptions, where the number of processors utilized to solve a problem is a function of problem size. It is these advantages that make the PRAM easy to program and popular for algorithm development. The success of the PRAM model is witnessed by the hundreds of algorithms that have been designed and extensively analyzed on various variants of the PRAM.

In parallel machines that have been built so far, however, communication is expensive, synchronization is not instantaneous and unlimited parallelism is unavailable. The majority of parallel machines and available parallel software do not allow programmers to write programs that are both efficient, portable and scalable; such objectives are only feasible by fine-tuning parallel code and taking into account the finest details of the underlying architecture and the programming interface that is available to the programmer. Parallel algorithm design, on the other hand, usually ignores communication and/or synchronization issues and works only under unlimited parallelism assumptions (or by simulations on limited processor models). Many parallel algorithms are thus designed having in mind factors not present in any reasonable machine, such as zero communication delay or infinite bandwidth.

The need for architecture independent parallel algorithm design is thus immediate for effective and efficient programming of existing and future parallel hardware platforms. Such an algorithm design would work in two steps. First, an abstraction of parallel hardware is obtained to allow the performance of a parallel machine to be described in some general architecture independent terms through a number of parameters that reflect the computation, communication, and synchronization capabilities of the hardware. Then, the performance of a parallel algorithm would be expressed as a function of these parameters and problem size similarly to traditional sequential algorithm design. This way it would be possible to describe the performance of a parallel algorithm not only on existing machines but also on machines not currently built as long as sufficient information on the not-yet-built machines can be obtained (i.e. the parameters of the abstraction become available).

The introduction of realistic parallel computer models such as the Bulk-Synchronous Parallel (BSP) model [65, 66] and the LogP [15] model and their extensions ([47, 7]) come to address these limitations of parallel computing. Both models take into reasonable consideration synchronization, communication, and latency issues related to both communication, bandwidth limitations and network conflicts during routing.

A considerable amount of work has been devoted in the study of the BSP model, and the analysis and design of BSP algorithms. In this work, we present parallel implementations, using the Oxford BSP Toolset, *BSPlib* [43], of deterministic and randomized BSP sorting algorithms on a distributed memory machine, a Cray T3D. A report is also included on the performance and relative merits of these algorithms and the knowledge gained from the study of the predicted theoretical performance and its comparison to actual running time. A comparison of our algorithm implementations to other parallel sorting algorithm implementations is also presented.

1.1 The Parallel Programming Model

The *Bulk-Synchronous Parallel* (BSP) model of computation deals explicitly with the notion of communication and synchronization among computational tasks and has been proposed in [65, 66] as a unified framework for the design, analysis and programming of general purpose parallel computing systems. It offers the prospect of achieving both scalable parallel performance and architecture independent portable and reusable parallel software and provides a framework which permits the performance of parallel software to be analyzed and predicted in a consistent and concise way. Processor components in the BSP model advance jointly through the program, with the required remote communication occurring between *supersteps*. A superstep may be thought of as a segment of computation during which each processor performs a given task using data local to the processor before the start of the superstep. Such a task may include local computations, message transmissions, and message receipts. A BSP computer as described in [64, 65, 66] then consists of the following three components: (a) a *collection of p processor/memory components*, (b) a *communication network* that delivers messages point to point among the components, and (c) *facilities* for synchronization of all or a subset of the processors.

Any BSP computer can be modeled by p , the number of processor components, L , the minimal time, measured in terms of basic computation steps, between successive synchronization operations, and g the ratio of the total throughput of the whole system in terms of basic computation steps, to the throughput of the router in terms of words of information delivered. The definition of g relates to the routing of an h -relation in continuous message usage, that is, the situation where each processor sends or receives at most h messages (or words of information); g is the cost of communication so that an h -relation is realized within gh steps [65], for any h such that $h \geq h_0$, where h_0 is a machine dependent parameter. Otherwise, if $h < h_0$, the cost of communication is L .

A superstep can complete at any time after L time units. The time complexity of a superstep S in a BSP algorithm is determined as follows. Each superstep is charged $\max\{L, x + gh\}$ basic time steps, where x is the maximum number of basic computational operations executed by any processor during S , and h is the maximum amount of information transmitted or received by any processor.

The performance of a BSP algorithm \mathcal{A} is specified in three parts. First, a sequential algorithm \mathcal{A}^* is specified with which the BSP algorithm is compared and the charging policy for basic operations of both \mathcal{A} and \mathcal{A}^* is made explicit. Ratios between runtimes on pairs of models that have the same set of local instructions will thus be measured and therefore operations can be defined in a higher level of abstraction than machine level instructions.

Second, two ratios π and μ are specified. The former, π , is the ratio between the computation time $C_{\mathcal{A}}$, of the BSP algorithm, over the time $C_{\mathcal{A}^*}$ of the comparing sequential algorithm divided by p , i.e., $\pi = pC_{\mathcal{A}}/C_{\mathcal{A}^*}$, and is a measure of the computational efficiency of \mathcal{A} . The latter, μ , is the ratio between the parallel time $M_{\mathcal{A}}$ required by the communication supersteps of the BSP algorithm and the computation time of \mathcal{A}^* divided by p , i.e., $\mu = pM_{\mathcal{A}}/C_{\mathcal{A}^*}$, and is a measure of the impact of communication in achieving optimal efficiency (the lower the μ the more communication efficient \mathcal{A} is). The ratio $p/(\pi + \mu)$ gives the speedup, and the ratio $1/(\pi + \mu)$ the parallel efficiency of algorithm \mathcal{A} .

Finally, conditions on n , p , L and g are then specified that are sufficient for the algorithm to make sense and the claimed (upper) bounds on π and μ to hold. Corollaries are claimed for some sufficient conditions for the most interesting optimality criteria, such as c -optimality, i.e., $\pi = c + o(1)$ and $\mu = o(1)$. Insistence on one-optimality leads to algorithms that only work for restricted ranges of p , L and g . All asymptotic bounds refer to the problem size as $n \rightarrow \infty$. The other parameters may also tend to ∞ if they

are expressed in terms of n , though it is not assumed so.

The charging policy for local operations on the BSP model for the sorting algorithms is described. Since the mechanisms of the BSP model are similar to those used by sequential models and performance ratios of these two are important, operations can be defined in a higher level than machine level instructions. A dominant term in the BSP algorithms to be described later is contributed by sequential sorting performed independently at each processor. A charge of $n \lg n$ time for sorting n keys sequentially [49], and $n \lg q$ time for merging q lists of total size n [49] are claimed. For other operations, $O(1)$ time is charged for operations over an associative operator in parallel-prefix computations and $O(1)$ time for each comparison performed. Finally, $\lceil \lg n \rceil$ comparisons are charged for performing binary search in a sorted sequence of length $n - 1$.

2 Sorting on the BSP model: An overview

A significant amount of work has been devoted in the study of the BSP model [64, 65, 66, 53, 54, 55, 47, 7] and the analysis [7, 10, 11, 21, 22, 36] and design of BSP algorithms [8, 9, 23, 24, 25, 26, 27, 53, 54, 55].

The problem of parallel sorting n keys [51] has drawn considerable attention. First approaches were sorting networks introduced by Batcher [5]. Since then, a wealth of literature concerning parallel sorting has been published [50, 51]. The first sorting network to achieve, however, the optimal $O(\lg n)$ time bound to sort n keys is the AKS [2] network of width $O(n)$. Subsequently, Reif and Valiant [59] presented a randomized sorting algorithm that runs on a fixed-connection network and achieves the same time bound as the AKS algorithm; the constant multipliers in the randomized algorithm are, however, considerably smaller. Cole [14] was the first to present optimal $O(\lg n)$ time sorting algorithms for n -processor CREW and EREW PRAMs. These methods fail, however, to deliver optimal performance when directly implemented on the BSP model of computation.

Note: In the expressions for running time on the BSP model below, terms that involve L are included in the communication – and not in computation – time, to simplify exposition. Polynomial slack would mean that $n_p = n/p$ is a constant degree polynomial, i.e. $p = n^{1-t}$ for some constant t such that $0 < t < 1$.

Previously known results on BSP sorting include deterministic [1, 10, 22, 27, 36, 62] and randomized [21, 24, 30] algorithms.

An adaptation of the AKS sorting network [2] on the BSP model is shown in [10] to yield computation and communication time $O(n_p \lg n)$ and $gO(n_p \lg p) + LO(\lg p)$ respectively. The constant factors hidden in this algorithm, however, is considerably large and not of practical significance.

It is shown in [1] that for all values of n and p such that $n \geq p$, a BSP version of column-sort [50] requires $O(\zeta^{3.42} n_p \lg n_p)$ and $O(g\zeta^{3.42} n_p) + O(L\zeta^{3.42})$ time for computation and communication respectively, where $\zeta = \lg n / \lg(n/p)$.

In [10], it is shown that an adaptation on the BSP model of the cube-sort algorithm [16] requires computation time $O(25^{\lg^* n - \lg^*(n_p)} \zeta^2 n_p \lg n_p)$ and $O(g 25^{\lg^* n - \lg^*(n_p)} \zeta^2 n_p) + O(L 25^{\lg^* n - \lg^*(n_p)} \zeta^2)$ for communication. Furthermore, as noted in [36], a modification of cube-sort shown in [58] eliminates the term $25^{\lg^* n - \lg^*(n_p)}$ and the resulting algorithm improves upon the algorithm of [1].

In [21], a randomized BSP sorting algorithm that uses the technique of oversampling is introduced that for an ample range of the parameters n and p , i.e., $p = O(n / \lg^{1+\alpha} n)$, for any constant $\alpha > 0$, requires – under realistic assumptions – computation and communication time $(1 + o(1)) (n \lg n / p)$ and $O(g \zeta n_p) + O(L \zeta)$ respectively, with high-probability. A similar algorithm for the case $p^2 < n$, but with a less tighter

analysis is discussed in [52]. Given any $O(n \lg n)$ worst-case running time sequential sorting algorithm, the randomized BSP algorithm utilizes the sequential algorithm for local sorting and exhibits parallel efficiency which is asymptotically 100% provided that the BSP parameters are appropriately bounded (and these bounds accommodate most, if not all, currently built parallel machines). The constants hidden affect low order terms and are well defined. This has been one of the first attempts to establish in some general way parallel efficiency for an algorithm by taking into consideration communication and synchronization delays and describing optimality criteria not in absolute terms, that may be meaningless, but by comparing the proposed parallel algorithm to the best sequential one. The bounds on processor imbalance during sorting are tighter than any other random sampling/oversampling algorithm [46, 20, 59, 60]. The randomized algorithm has been implemented on realistic machines (SGI Power Challenge, Cray T3D and IBM SP2) and exhibited performance well within the bounds indicated by the theoretical analysis ([4, 31]). We note that sample-based sorting algorithms are extensively studied and implemented in the literature [12, 42]. For example, the sample-sort algorithm in [12] is computationally equivalent to the one round case of [21]. The analysis of [21] is however much tighter. It allows better control of oversampling with the end result of achieving smaller bucket expansion (key imbalance during the routing operation). The work of [42] applied to d -dimensional meshes is similar to the general algorithm of [21].

In [27, 30], it is shown that asymptotical performance comparable to that of [21] can be achieved for values of p much closer to n , i.e., $p = \omega(n \lg \lg n / \lg n)$ by a new randomized sorting algorithm. The failure probability is, however, significantly lower. As the improvements of this algorithm are interesting only for large p and the corresponding conditions on L and g for large p can be satisfied only by machines with very low L and g (i.e. PRAM-like behaving machines), the practical implications of this algorithm are less significant and interesting than its theoretical merit.

Under the same realistic assumptions as [21], a new deterministic sorting algorithm is introduced in [22], and a bound of $(1 + (\lfloor \zeta(1 - \zeta^{-1}) \rfloor)((1 - \zeta^{-1})/2) + o(1))(n \lg n/p)$ and $O(g\zeta n_p) + O(L \zeta)$ is shown for computation and communication respectively, thus improving upon the upper bounds of [1, 2, 10, 16, 50]. The bound on computation is subsequently improved in [27, 28]. In particular, for $p = n / \lg^{2+\alpha} n$, $\alpha = \Omega(1)$, the improved deterministic sorting algorithm in [27, 28] requires computation and communication time $(1 + 2/\alpha + o(1)) n \lg n/p$ and $O(g \zeta n_p) + O(L \zeta)$ respectively. The algorithm performs deterministic sorting by extending regular sampling, a technique introduced in [61], to perform deterministic regular oversampling. Past results that use regular sampling have been available for cases with $p^2 < n$. The BSP algorithm in [22, 27, 28] further extends the processor range and achieves asymptotically optimal efficiency for a range of n/p that is very close to that ratio of the randomized BSP algorithms in [21, 30]. This was made possible by a detailed and precise quantification of the input key imbalance among the processors during the phases of deterministic sorting thus contributing to the understanding of regular oversampling. By using regular oversampling in a deterministic setting, it is possible to regulate the oversampling ratio to bound the maximum key imbalance among the processors. As in the case of randomized sorting, the insistence, under an architecture independent algorithm design, of satisfying the criterion of *one-optimality* (i.e. optimal speedup/efficiency) led to these improvements. In this work we use the structure of this algorithm to derive a sample-based randomized sorting algorithm that does not follow the standard pattern of sample and splitter select, key routing and local sorting but instead follows the pattern of the deterministic algorithm i.e. local sort, sample and splitter select, key routing and local merging.

In [36, 62] independent BSP adaptations of parallel merge-sort [14] are presented that for all values of n and p such that $p \leq n$, require computation and communication/synchronization time $O(n \lg n/p)$

and $O(g\zeta n_p) + O(L\zeta)$ respectively. As the BSP adaptation of parallel merge-sort is at least as involved as the original parallel merge-sort (EREW PRAM variant) algorithm [14], the constant factors involved in the analysis are considerably large and the algorithm seems to be of little practical use as opposed to the algorithms in [21, 22, 28].

3 Contents of the paper

In this work we study an implementation of the deterministic algorithm in [27, 28] that uses deterministic regular oversampling, which extends the notion of regular sampling of [61], and parallel sample sorting that allows the algorithm to work for a wider range of processor size p . The deterministic algorithm is also augmented to handle transparently duplicate keys with optimal asymptotic efficiency. Our method of duplicate-key handling tags only a fraction $o(n)$ of the n input keys, and does not double communication and/or computation time that other duplicate handling approaches may require [39, 40, 41]. The idea of using deterministic regular-oversampling in a deterministic sorting algorithm and the accurate quantification of its effects by measuring constants under the BSP model, results in a fine and quantifiable balance of computational load and communication time. Combined with our transparent and efficient duplicate key handling it leads to an algorithm that maintains its optimal performance even if all keys are the same.

The basis of the randomized algorithm is the foundation of randomized sample-sort algorithms [46, 20, 59, 60] that use oversampling [59] and in particular the BSP adaptation discussed in [21] combined with the ideas used in the deterministic algorithm. The resulting randomized sorting algorithm whose implementation is studied works differently from other sample-sort based algorithms including the one in [21]. The algorithm first local sorts, then sample and splitter selects, routes keys, and in the very end p -way merges sorted sequences as opposed to traditional sample-sort algorithms that sample and splitter select, route keys, and local sort. Our approach augmented with the transparent and efficient duplicate-key handling also used in the deterministic algorithm results in improved parallel performance, work balance and communication minimization.

Even though the two algorithms look similar, randomized oversampling is provably superior to deterministic regular-oversampling. The oversampling parameter in the former case can vary more widely than the corresponding one of the latter case thus resulting in more balanced communication and balanced work-load among the processors.

Parallel sorting algorithms originally designed for other models of computation (e.g. bitonic-sort [5, 29]) are also implemented. The implementation platform is a distributed memory machine, a Cray T3D, and the communication library used is the Oxford BSP Toolset [43].

The introduced algorithms are designed and analyzed in an architecture independent setting. Such a design is helpful in deciding the best way to implement various parallel operations optimally on the given platform based on the knowledge of the BSP parameters of the target platform only. For example, problem size n of the experiments, and machine configuration parameters as expressed in terms of the BSP parameters p , L and g determine the choice of broadcasting and parallel prefix algorithms in parallel sorting. It is also used in determining the choice of values of certain parameters of the algorithm implementations (such as those related to oversampling issues). This choice of parameter values subsequently affects the maximum input key imbalance among the processors which has a significant impact on running time. Insistence on one-optimality both in the design and analysis of the given algorithms is hoped to lead to efficient implementations that would confirm the claims of the theoretical analysis.

The theoretical analysis of the algorithms can be used in assessing the performance of the implementations because the designed algorithms have been shown to be one-optimal and this way, they have been directly compared to the best sequential implementations. In addition, in the theoretical analysis no hidden constants are involved with the most significant terms of parallel running time and the constants in low order terms are well understood. At the conclusion of the experiments theoretical performance is compared to observed performance on the test platform for this purpose.

In Section 4 we introduce some primitive operations that will be used in the sorting algorithms. In Section 5 we introduce the deterministic algorithm `Sort_Det_BSP` that will be implemented. The algorithm is one of several cases of a more general algorithm described in [28]. The performance of the implemented algorithm is however analyzed in detail and more accurately than the general algorithm of [28], and experiments on its implementation are discussed in Section 6. Its augmentation to handle duplicate keys transparently and with optimal asymptotic efficiency is also discussed separately in Section 5.1.1. Subsequently, a special case of the randomized BSP algorithm in [21] is introduced, called `Sort_Ran_BSP` that is a sample-sort based algorithm. The randomized algorithm of our implementations, `Sort_Iran_BSP`, derives from `Sort_Det_BSP` and `Sort_Ran_BSP` and is introduced in the same section; it works differently however from traditional sample-sort based algorithms. Duplicate-key handling is also implemented transparently for `Sort_Iran_BSP` using the technique of Section 5.1.1 used for `Sort_Det_BSP`. For each of the two implemented algorithms, two variants are studied experimentally depending on how sequential sorting is performed. Finally the obtained experimental results are discussed in detail in Section 6 and compared to other parallel sorting implementations like those in [44, 39, 40, 41].

The results we obtained justify the belief that architecture independent parallel algorithm design and analysis is possible, plausible, reliable and consistent and can be used to model and predict the performance of parallel algorithms on a variety of parallel machines with satisfactory accuracy. The vehicle for such an architecture independent parallel algorithm design and analysis has been the BSP model of computation. Its parameters seem to model a parallel computer well enough to make parallel program performance estimation plausible for such diverse problems as matrix computations ([34]) and parallel sorting. The experiences related to the latter problem are described in more detail in this work.

4 Primitive Operations

In this section BSP algorithms for two fundamental operations, namely *broadcast* and *parallel-prefix* and *parallel radix-sort* are introduced. All these primitives are auxiliary routines for the algorithms described in later sections. A fully detailed account of such operations along with additional results can be found in [26].

Lemma 4.1 *There exists a BSP algorithm for broadcasting an n -word message that requires time at most $M_{brd}^n(p) = (\lceil n/\lceil n/h \rceil \rceil + h - 1) \max \{L, gt\lceil n/h \rceil\}$, for any integer $2 \leq t \leq p$, where $h = \lceil \log_t ((t-1)p + 1) \rceil - 1$.*

Proof: The underlying algorithm employs a pipelined t -ary tree, consisting of p nodes, for some appropriate t . The depth of the tree is $h = \lceil \log_t ((t-1)p + 1) \rceil - 1$. In each superstep, the root processor of a subtree sends t copies of a separate m -word segment of the original message to its children, where $m = \lceil n/h \rceil$. Similarly, each internal processor sends t copies of the message that it received in the previous superstep to its own children. The algorithm completes within at most $\lceil n/m \rceil + h - 1$ supersteps. Each

superstep requires gtm communication time, and the lemma follows. ■

Lemma 4.2 *There exists a BSP algorithm for computing n independent parallel-prefix operations that requires time at most $C_{ppf}^n(p) = 2(\lceil n/\lceil n/h \rceil \rceil + h - 1) \max\{L, t\lceil n/h \rceil\}$ and $M_{ppf}^n(p) = 2(\lceil n/\lceil n/h \rceil \rceil + h - 1) \max\{L, g2t\lceil n/h \rceil\}$, for computation and communication respectively, for any integer $2 \leq t \leq p$, where $h = \lceil \log_t p \rceil$, for a total time of $T_{ppf}^n(p) = C_{ppf}^n(p) + M_{ppf}^n(p)$.*

Proof: The underlying algorithm consists of two passes of the algorithm implied by Lemma 4.1, on a pipelined t -ary tree, consisting of p leaf nodes and at most p internal nodes, for some appropriate t . The depth of the tree is $h = \lceil \log_t p \rceil$. The lemma follows by way of arguments similar to that of Lemma 4.1. ■

For $n = 1$, $T_{ppf}(p)$ (respectively $C_{ppf}(p)$, $M_{ppf}(p)$) can be written for $T_{ppf}^1(p)$ (respectively $C_{ppf}^1(p)$, $M_{ppf}^1(p)$). The same notational convention applies to all pipelined operations.

5 The Algorithms

5.1 BSP Deterministic Sorting Algorithm in [22, 28]

The deterministic algorithm of the implementations is based on a non-iterative variant of the sorting algorithm of [22, 27, 28] which has been shown to be one-optimal for a satisfactory range of the BSP parameters that includes most currently built parallel machines; for a wider range of these parameters the algorithm is c -optimal, where $c \geq 1$ is a small constant. The algorithm is regular-sampling based ([61]) but extends regular sampling to regular oversampling and utilizes an efficient partitioning scheme that splits – almost evenly and independently of the input distribution – an arbitrary number of sorted sequences among the processors. In Section 5.1.1 this base algorithm is augmented to handle transparently and in optimal asymptotic efficiency duplicate keys. In our approach duplicate handling does not require doubling of communication and/or computation time that other approaches seem to require [39, 40, 41]. The base algorithm consists of the following phases:

- (1) *Local Sorting.* Each processor, in parallel with all the other processors, sorts its local input sequence of size n/p that resides in its local memory.
- (2) *Partitioning.* Processors cooperate to evenly split the sorted sequences among the processors.
- (3) *Merging.* Each processor, in parallel with all the other processors, merges a small number of subsequences of total size $(1 + o(1))(n/p)$.

The iterative version of the algorithm performs m iterations of phases 2 and 3. In each iteration the data set is partitioned into k buckets of approximately equal size. In the following iteration a similar process of sub-partitioning each of these k buckets into k further buckets is performed, and so on. The maximum number of keys per processor in any of the m iterations can be shown to be $(1 + O(1/\omega_n))(n/p)$. By employing multi-way merging [49] the desired sorted sequence can be obtained.

In Figure 1, function `SORT_DET_BSP(X)`, implements the sorting operation for $m = 1$. X denotes the input sequence, n the size of X , p the number of processors, and s is a user defined parameter (oversampling factor) that inversely relates to the maximum possible imbalance of the sequences formed in step (13) of the algorithm. For any sequence X , the subsequence of X residing in processor k is denoted $X^{(k)}$.

The implemented version `SORT_DET_BSP` as a special case of the more general algorithm reported in [22, 27, 28] can also be viewed as an adaptation of the regular-sampling algorithm of [61] on the BSP model. It differs, however, from the algorithm in [61] in that sample-sorting is performed in parallel and deterministic oversampling is used with the purpose of achieving finer load balancing in communication. The choices in deterministic oversampling that also affect load-balancing in the subsequent key routing step are based on the quantifiable results obtained from the detailed analysis of [28]; although oversampling has been used previously in randomized sorting, its usefulness in deterministic sorting was not explored and quantified in full.

```

begin SORT_DET_BSP ( $X$ )
1.  let  $r = \lceil \omega_n \rceil$  ;
2.  for each processor  $\langle k \rangle$ ,  $k \in \{0, \dots, p-1\}$ , in parallel
3.    do SORT_SEQ( $X^{(k)}$ ) ;
4.    form locally a sample  $Y^{(k)} = \langle y_1, \dots, y_{rp-1} \rangle$  of  $rp-1$  evenly spaced
        keys that partition  $X^{(k)}$  into  $s = rp$  evenly sized segments
        and append the maximum of  $X^{(k)}$  to this sequence ;
5.  let  $\bar{Y} = \text{BITONIC\_SORT}(Y)$  ;
6.  form  $S = \langle s_s, \dots, s_{(p-1)s} \rangle$  from  $\bar{Y}$  that consists of  $p-1$  evenly spaced splitters
        that partition  $\bar{Y}$  into  $p$  evenly sized segments ;
7.  BROADCAST ( $S$ );
8.  for each processor  $\langle k \rangle$ ,  $k \in \{0, \dots, p-1\}$ , in parallel
9.    do partition  $X^{(k)}$  into  $p$  subsequences  $X_0^{(k)}, \dots, X_{p-1}^{(k)}$  as induced by the
         $p-1$  splitters of  $S$  ;
10.   for all  $i \in \{0, \dots, p-1\}$ 
11.     do communicate subsequence  $X_i^{(k)}$  to  $Z_k^{(i)}$  ;
12.   let  $\bar{X}^{(k)} = \text{MERGE\_SEQ}(\bigcup_i Z_i^{(k)})$  ;
13.   return  $\bar{X}^{(k)}$  ;
end SORT_DET_BSP

```

Figure 1: Procedure `SORT_DET_BSP`.

The proposition and proof below simplify the results shown in a more general context in [22, 27, 28].

Proposition 5.1 *For any n and $p \leq n$, and any function ω_n of n such that $\omega_n = \Omega(1)$, $\omega_n = O(\lg n)$ and $p^2 \omega_n^2 \leq n / \lg n$, and for $n_{\max} = (1 + 1/\lceil \omega_n \rceil)n/p + \lceil \omega_n \rceil p$, algorithm `SORT_DET_BSP` requires time, $(n/p) \lg(n/p) + n_{\max} \lg p + O(p + \omega_n p \lg^2 p)$ for computation and $gn_{\max} + L \lg^2 p/2 + O(L + g\omega_n p \lg^2 p)$ for communication.*

Corollary 5.1 *For n , p and ω_n as in Proposition 5.1, algorithm `SORT_DET_BSP` is such that $\pi = 1 + \lg p / (\lceil \omega_n \rceil \lg n) + O(1/(\omega_n \lg n) + \lg^2 p / (\omega_n \lg^2 n))$, and for $L \leq 2n/(p \lg^2 p)$, $\mu = (1 + 1/\lceil \omega_n \rceil)g / \lg n + Lp \lg^2 p / (2n \lg n) + O(g \lg^2 p / (\omega_n \lg^2 n) + 1/(\lg n \lg^2 p))$ as well.*

Proof: The input is assumed to be evenly but otherwise arbitrarily distributed among the p processors before the beginning of the execution of the algorithm. Moreover, the keys are distinct since in an extreme case, we can always make them so by, for example, appending to them the code for their memory location. We later explain how we handle duplicate keys without doubling (in the worst case) the number of comparisons performed. Parameter ω_n determines the desired upper bound in processor key imbalance during

the key routing operation. The term $1 + 1/\lceil \omega_n \rceil$ is also referred to as bucket expansion in sample-sort based randomized sorting algorithms ([12]).

In step 3, each processor sorts the keys in its possession. As each processor holds at most $\lceil n/p \rceil$ keys, this step requires time $\lceil n/p \rceil \lg \lceil n/p \rceil$. Algorithm SORT_SEQ is any sequential sorting algorithm of such performance.

Subsequently, each processor selects locally $\lceil \omega_n \rceil p - 1 = rp - 1$ evenly spaced sample keys, that partition its input into $\lceil \omega_n \rceil p$ evenly sized segments. Additionally, each processor appends to this sorted list the largest key in its input. Let $s = \lceil \omega_n \rceil p = rp$ be the size of the so identified list. Therefore step 4 requires time $O(s)$.

The p sorted lists, each consisting of s sample keys, are merged; let sequence $\langle s_1, s_2, \dots, s_{ps} \rangle$ be the result of the merge operation. By assumption, the sequence is evenly distributed among the p processors, i.e., subsequence $\langle s_{is+1}, \dots, s_{(i+1)s} \rangle$, $0 \leq i \leq p - 1$, resides in the local memory of the i -th processor. The cost of step 5 is that of parallel sorting by one of Batcher's methods [5], appropriately modified [49] to handle sorted sequences of size s . The computation and communication time required for this stage is, respectively, $2s(\lg^2 p + \lg p)/2$ and $(\lg^2 p + \lg p)(L + gs)/2$.

In step 6, a set of evenly spaced splitters is formed from the sorted sample. A broadcast operation is initiated in step 7, where splitter s_{is} , $1 \leq i < p$, along with its index in the sequence of sample keys is sent to all processors. Lines 6 and 7 require time $O(1)$ and $\max\{L, gO(p)\} + M_{brd}^{p-1}(p)$ for computation and communication respectively.

In step 9, each processor decides the position of every key it holds with respect to the $p - 1$ splitters it received in step 7, by way of sequential merging the splitters with the input keys in $p - 1 + n/p$ time or alternately by performing a binary search of the splitters into the sorted keys in time $p \lg(n/p)$, and subsequently counts the number of keys that fall into each of the p so identified buckets induced by the $p - 1$ splitters. Subsequently, p independent parallel prefix operations are initiated (one for each subsequence) to determine how to split the keys of each bucket as evenly as possible among the processors using the information collected in the merging operation. The p disjoint parallel prefix operations in step 9 are realized by employing the algorithm of Lemma 4.2 thus resulting in a time bound of $p \lg(n/p) + T_{ppf}^p(p)$ for step 9.

In step 11, each processor uses the information collected by the parallel prefix operation to perform the routing in such a way that the initial ordering of the keys is preserved (i.e. keys received from processor i are stored before those received from j , $i < j$, and also the ordering within i and j is also preserved). Step 11 takes time $\max\{L, gn_{max}\}$.

In step 12, each processor merges the at most p sorted subsequences that it received in step 11. When this step is executed, each processor, by way of Lemma 5.1 to be shown, possesses at most $p = \min\{p, n_{max}\}$ sorted sequences for a total of at most n_{max} keys, where $n_{max} = (1 + 1/\lceil \omega_n \rceil)(n/p) + \lceil \omega_n \rceil p$. The cost of this stage is that of sequential multi-way merging n_{max} keys by some deterministic algorithm [49], which is $n_{max} \lg p$, as $\omega_n^2 p = O(n/p)$.

Summing up all the terms for computation and communication and noting the conditions on L and g and assigning a cost of $n \lg n$ to the best sequential algorithm for sorting the result follows. ■

It remains to prove that at the completion of step 9 the input keys are partitioned into (almost) evenly sized subsequences. The main result is summarized in the following lemma.

Lemma 5.1 *The maximum number of keys n_{max} per processor in SORT_DET_BSP is $(1 + 1/\lceil\omega_n\rceil)(n/p) + \lceil\omega_n\rceil p$, for any ω_n such that $\omega_n = \Omega(1)$ and $\omega_n = O(\lg n)$, provided that $\omega_n^2 p = O(n/p)$ is also satisfied.*

Proof: Although it is not explicitly mentioned in the description of algorithm SORT_DET_BSP we assume that we initially pad the input so that each processor owns exactly $\lceil n/p \rceil$ keys. At most one key is added to each processor (the maximum key can be such a choice). Before performing the sample selection operation, we also pad the input so that afterwards, all segments have the same number of keys that is, $x = \lceil \lceil n/p \rceil / s \rceil$. The padding operation requires time at most $O(s)$, which is within the lower order terms of the analysis of Proposition 5.1, and therefore, does not affect the asymptotic complexity of the algorithm. We note that padding operations introduce duplicate keys; a discussion of duplicate handling follows this proof.

Consider an arbitrary splitter s_{is} , where $1 \leq i < p$. There are at least isx keys which are not larger than s_{is} , since there are is segments each of size x whose keys are not larger than s_{is} . Likewise, there are at least $(ps - is - p + 1)x$ keys which are not smaller than s_{is} , since there are $ps - is - p + 1$ segments each of size x whose keys are not smaller than s_{is} . Thus, by noting that the total number of keys has been increased (by way of padding operations) from n to psx , the number of keys b_i that are smaller than s_{is} is bounded as follows.

$$isx \leq b_i \leq psx - (ps - is - p + 1)x.$$

A similar bound can be obtained for b_{i+1} . Substituting $s = \lceil\omega_n\rceil p$ we therefore conclude the following.

$$b_{i+1} - b_i \leq sx + px - x \leq sx + px = \lceil\omega_n\rceil px + px.$$

The difference $n_i = b_{i+1} - b_i$ is independent of i and gives the maximum number of keys per split sequence. Considering that $x \leq (n + ps)/(ps)$ and substituting $s = \lceil\omega_n\rceil p$, the following bound is derived.

$$n_{max} = \left(1 + \frac{1}{\lceil\omega_n\rceil}\right) \frac{n + ps}{p}.$$

By substituting in the nominator of the previous expression $s = \lceil\omega_n\rceil p$, we conclude that the maximum number of keys n_{max} per processor of function SORT_DET_BSP is bounded above as follows.

$$n_{max} = \left(1 + \frac{1}{\lceil\omega_n\rceil}\right) \frac{n}{p} + \lceil\omega_n\rceil p.$$

The lemma follows. ■

The particular algorithm as depicted in Figure 1 corresponds to the simplest case of the deterministic algorithm in [28] where an analysis for all possible values of p is presented. In the general algorithm, the number of communication rounds is $\lg n / \lg(n/p)$ for any $p = n^{1-t}$, with t constant, i.e. it is constant. If, however, t is not constant, then the number of rounds becomes $O(\lg n / \lg(n/p))$ still matching in all cases the lower bound of [36]. We note that in the general case, the implementation of parallel prefix and broadcasting operations depends on p , L , g and the amount of information processed by these operations. This highlights a difference between *architecture independent parallel* algorithm design and *classic parallel* algorithm design. For a given choice of problem size n , and machine i.e. for a given tuple (n, p, L, g) , the resulting algorithm may differ from that chosen for another tuple (n', p', L', g') . In the case of sorting for example, one algorithm may implement sample sorting sequentially while another one in parallel, one algorithm may implement a parallel prefix or broadcasting operation using a PRAM approach in $\lg p$ supersteps while another algorithm may perform the same operations in constant number of supersteps as in Lemma 4.1 or 4.2.

5.1.1 Duplicate-key Handling

Algorithm `SORT_DET_BSP`, as described, does not handle duplicate keys. A naive way to handle duplicate keys is by making the keys distinct. This could be achieved by attaching to each key the address of the memory location it is stored in. For data types whose bit or byte length is comparable to the length of the address describing them, such a transformation leads – in most cases – to a doubling of the overall number of comparisons performed and the communication time in the worst case. For more complex data types such as strings of characters the extra cost may be negligible.

An alternative way to handle duplicate keys is the following one that was also used in the implementations and handles duplicate keys in a transparent way that provides asymptotic optimal efficiency and tags only a small fraction of the keys. This seems to be an improvement over other approaches [39, 40, 41] that require a doubling of communication time. Procedure `SORT_SEQ` is implemented by means of a stable sequential sorting algorithm. Two tags for each input key are already implicitly available by default, and no extra memory is required to access them. These are the processor identifier that stores a particular input key and the index of the key in the local array that stores it. No additional space is required for the maintenance of this tagging. In our duplicate-key handling method such tags are only used for sample and splitter-related activity.

As sample sorting is a global operation, for sample sorting and splitter distribution only we augment every sample key into a record that includes this additional tag information (array index and processor storing the key). As the additional tagging information affects the sample only, and the sample is $o(1)$ of the input keys, the memory overhead incurred is small, as is the computational overhead. The attached tag information is used in step 4 to form the sample, in step 5 for sample sorting, in steps 6 and 7 for splitter selection and broadcasting, and finally in step 9 as all these steps require distinct keys to achieve stability and load-balance. In step 9 in particular, a binary search operation of a splitter key into the locally sorted keys involves first a comparison of the two keys. If the keys are equal the result of the comparison is resolved by comparing the readily (and implicitly) available identifier of the processor that holds the local key to the processor storing the comparing splitter (available through the tagging in step 4, and the broadcasting of step 7). If the two processor identifiers are equal, then the result of the comparison is determined by comparing the indices of the position in the local array that stores the local key and the splitter being compared.

In addition, the merging operation must also be performed in a stable manner, that is if, the keys at the head of two sorted sequences are equal the one received from processor i appears before the one received from processor j , $i < j$ in the output sequence of the operation.

The computation and communication overhead of duplicate handling that is described by this method is within the lower order terms of the analysis and therefore, the optimality claims still hold unchanged. The results on key imbalance still hold as well. This same duplicate handling method is also used in the implementation of algorithm `SORT_IRAN_BSP`.

5.2 BSP Randomized Sorting Algorithm in [21]

Randomized BSP sorting was introduced in [21]. An architecture independent analysis of the algorithm in [21] in terms of problem size n and p, L and g shows that it is one-optimal for most cases of interest. The algorithm derives from quicksort, the ideas of [20, 60] and the technique of oversampling [59]. It achieves the claimed efficiency as follows.

(1). The algorithm satisfies the requirement of one-optimality by using oversampling. The oversampling factor is in general $\omega(\lg n)$ and a practical choice that is being used in the experiments is $\Theta(\lg^2 n)$.

(2). As the size of the sample is smaller than input size n and parallel sampling is used, sample sorting can be performed either sequentially (by sending the sample to a single processor and sorting locally) or, as noted in [21], recursively, or by employing a non-optimal but in practice faster parallel algorithm such as Batcher's bitonic or odd-even merge sort as any of the latter two is simpler to implement and incurs low overhead costs for small problem instances.

(3). At every recursive call of classic quicksort an input sequence is split into two subsequences; a naive parallel implementation of quick sort would require in the best case $\lg n$ communication rounds, one for each recursive call, each round requiring $O(gn/p)$ time for communication. At the end of the $\lg p$ -th round, the input is split into p segments which is equal to the number of available processors. In a well-designed parallel implementation of quick sort, communication is only incurred in the first $\lg p$ rounds for a total of $O(gn \lg p/p)$. Total computation time is, however, $O(n \lg n/p)$. If p polynomially related to n , i.e. $p = n^{1-t}$, where $0 < t < 1$ is a constant, such an implementation is still inefficient as communication time is of the order of computation time even for $g = O(1)$.

(4). Based on ideas of [20] it then makes sense to split the input sequence into k sets, where $k > 2$, by choosing $k - 1$ splitters at every phase. This way the number of communication rounds is reduced to $\lg p / \lg k$. For $p = n^{1-t}$ as before, by choosing k so that $k = p^t$, the number of communication rounds is then $\lg n / \lg(n/p)$. For constant t , this is constant and therefore, communication time is smaller ($O(gn/p)$) than before. This observation is used in the randomized algorithm of [21] that describes the various cases of interest: (a) $p \leq \sqrt{n}$ that is of practical interest, (b) $p = n^{1-t}$, where t is constant and the algorithm is still interesting in terms of its practical implications and (c) for all other cases, p can grow as large as $p = O(n / \lg^{1+a} n)$, for any constant $a > 0$ and one-optimality can still be maintained. For most practical applications the number m of communication rounds is one (case (a)) or in some extreme cases at most 2 (case (b)). Which of the algorithms in (a) or (b) applies depends on the values of n , p , L and g .

Although partitioning and oversampling in the context of sorting are well established techniques, the analysis in [21] summarized in Claim 5.1 below allows one to prove the one-optimality of the sorting algorithm by quantifying precisely the key imbalance among the processors. Let $X = \langle x_1, x_2, \dots, x_N \rangle$ be an ordered sequence of keys indexed such that $x_i < x_{i+1}$, for all $1 \leq i \leq N - 1$. The implicit assumption is that keys are unique. Let $Y = \{y_1, y_2, \dots, y_{ks-1}\}$ be a randomly chosen subset of $ks - 1 \leq N$ keys of X also indexed such that $y_i < y_{i+1}$, for all $1 \leq i \leq ks - 2$, for some positive integers k and s . Having randomly selected set Y , a partitioning of $X - Y$ into k subsets, X_0, X_1, \dots, X_{k-1} takes place. The following result shown in [21] is independent of the distribution of the input keys.

Claim 5.1 *Let $k \geq 2$, $s \geq 1$, $ks < N/2$, $n \geq 1$, $0 < \varepsilon < 1$, $\rho > 0$, and*

$$s \geq \frac{1 + \varepsilon}{\varepsilon^2} \left(2\rho \log n + \log(2\pi k^2(ks - 1)e^{1/(3(ks-1))}) \right).$$

Then the probability that any one of the X_i , for all i , $0 \leq i \leq k-1$, is of size more than $\lceil (1+\varepsilon)(N-k+1)/k \rceil$ is at most $n^{-\rho}$.

Algorithm SORT_RAN_BSP describes case (4)(a) of [21], and this special case is widely referred to as sample-sort in the literature. X denotes the input key sequence, n the size of X , p the number of processors, and s is a user defined parameter (oversampling factor) that inversely relates to the maximum key imbalance of the split sequences formed in step (13) of the algorithm.

```

begin SORT_RAN_BSP ( $X$ )
1.  let  $s = 2\omega_n^2 \lg n$  ;
2.  select uniformly at random a sample  $Y = \langle y_1, \dots, y_{sp-1} \rangle$  of  $sp - 1$  keys ;
3.  communicate  $Y$  to processor  $\langle 0 \rangle$  ;
4.  if processor  $\langle 0 \rangle$  then
5.    do let  $\bar{Y} = \text{SORT\_SEQ}(Y)$  ;
6.    form locally a set  $S = \langle s_1, \dots, s_{p-1} \rangle$  of  $p - 1$  evenly spaced splitters
      that partition  $\bar{Y}$  into  $p$  evenly sized segments ;
7.    BROADCAST( $S$ ) ;
8.  for each processor  $\langle k \rangle$ ,  $k \in \{0, \dots, p - 1\}$ , in parallel
9.    do partition  $X^{(k)}$  into  $p$  subsets  $X_0^{(k)}, \dots, X_{p-1}^{(k)}$  as induced by the  $p - 1$ 
      splitters of  $S$  ;
10.   for all  $i \in \{0, \dots, p - 1\}$ 
11.     do communicate subset  $X_i^{(k)}$  to  $Z_k^{(i)}$  ;
12.   let  $\bar{X}^{(k)} = \text{SORT\_SEQ}(\bigcup_i Z_i^{(k)})$  ;
13.   return  $\bar{X}^{(k)}$  ;
end SORT_RAN_BSP

```

Figure 2: Procedure SORT_RAN_BSP.

Proposition 5.2 *For any ω_n, n, p such that $2p\omega_n^2 \lg p < n/2$, $p^2 \leq n$, and $L = o(n/(p \lg^2 p))$, algorithm SORT_RAN_BSP has $\pi = 1 + 1/\omega_n + 1/\lg n + 2p^2\omega_n^2 \lg p/n + o(1/\lg n)$ and $\mu = O(gp^2\omega_n^2/n) + O(g/\lg n) + o(g/\lg n)$.*

Proof: For the sake of completeness we use elements of the proof of [21] to illustrate the performance of the algorithm for the specific choice of splitters $k = p - 1$.

Step 2 of the algorithm requires parallel time $O(L + gsp)$ per processor. As shown in [21], processors select uniformly at random processor identifiers $0 \dots p - 1$ and send these identifiers to the identified processors. Chernoff bound techniques can be used to show that each processor sends or receives $O(s)$ identifiers for communication time $O(L + gs)$. Processors then select uniformly at random and without replacement a number of keys equal to the number of identifiers received previously, a step that takes $O(s)$ parallel time. Since all sample keys are then communicated to processor 0 this step would require $O(L + gsp)$ time. Step 5 requires time $sp \lg(sp)$ and splitter selection in step 6 takes $O(p)$ time.

The broadcasting in step 7 takes $O(L + gp)$ time as it will take place in one superstep. Each processor k then performs a binary search of its set $X^{(k)}$ of n/p input keys onto the $p - 1$ splitters to determine sets $X_0^{(k)} \dots X_{p-1}^{(k)}$. This requires at most $(n/p)(\lg p + 1)$ comparisons. By claim 5.1, with probability $1 - o(1)$, each of $\bar{X}^{(k)}$ is of size at most $(1 + 1/\omega_n)n/p$. Therefore, in step 11, each processor sends n/p and receives at most $(1 + 1/\omega_n)n/p$ keys for communication time $O(L + g(1 + 1/\omega_n)n/p)$ and computation time in step 12 of $(1 + 1/\omega_n)n/p \lg(n/p) + o(n/p)$ for local sorting, as $\ln(1 + x) \leq x$, $x < 1$.

The total parallel computation time of SORT_RAN_BSP is $(1 + 1/\omega_n)n \lg n/p + (n/p) + sp \lg(sp) + o(n/p) + O(L + p)$ and communication time is $O(gn/p + gps + L)$. If we compare the parallel algorithm to the best sequential algorithm that requires $n \lg n$ comparisons and noting that $L = o(n/(p \lg^2 p))$, the claimed bounds on π and μ are derived. ■

SORT_RAN_BSP maintains an oversampling factor $2\omega_n^2 \lg n$ that is $\Omega(\lg n)$.

An implementation of SORT_RAN_BSP of Figure 2 is straightforward except perhaps that of step 9. In step 9, set $X^{(k)}$ is split into p sets such that the keys of the i -th set are routed to processor i ; the sets are determined by a binary search operation of each key into the set of $p - 1$ splitters. The formation of the p sets in step 9 is equivalent to an integer sort operation with key the result of the binary search operation (i.e. destination processor). Such set formation operation is thus of linear time Dn/p ; constant D is significant however since it includes the cost of copying keys in memory so that keys with the same destination processor are stored in contiguous memory locations. For relatively small values of n , constant D has the same growth as $\lg n$, and therefore, asymptotic claims may not be valid. In addition it seems that the sorting operation of step 5 could be done in parallel rather than sequentially.

These observations were taken into consideration in the design of the randomized BSP sorting algorithm of the implementations so that its performance is fully optimized. The resulting algorithm is SORT_IRAN_BSP. It is this algorithm that was implemented rather than sample-sort SORT_IRAN_BSP. Note that SORT_IRAN_BSP differs from most traditional sample-sort randomized parallel sorting algorithms in that it follows the pattern of local sorting, sample and splitter selection, key routing and local merging rather than the traditional one of sample and splitter selection, key routing and local sorting that identifies traditional sample-sort based randomized parallel sorting algorithms. In particular, in the design and implementation of SORT_IRAN_BSP we addressed the first limitation related to step 9 of SORT_RAN_BSP. To this end we employed ideas from the deterministic algorithm [22, 27] described in SORT_DET_BSP. In particular, we sorted the input keys before realizing the communication in step (11) and before we performed the sampling operation. This requires the replacement of the sequential sorting algorithm in step (12) of SORT_RAN_BSP by a multi-way merging algorithm as the received sequences from at most $p - 1$ processors are already sorted. A binary search in step (9) is not required any more as we can merge the local sorted keys with the sorted splitters or perform a binary search of the splitters into the keys an operation that allows for coarse-grained communication in step (11). The resulting algorithm looks similar to SORT_DET_BSP and thus, attains performance at least comparable to that of the corresponding deterministic algorithm. Random sampling, however, gives the programmer more freedom to determine processor imbalance (ω_n in the deterministic case is $O(\lg n)$; there is no such limitation in the randomized case). For handling duplicate keys we employed the method of the deterministic algorithm implementation described in Section 5.1.1 which effectively tags few keys only (sample keys) and thus uses asymptotically as much memory as the non-duplicate handling variant. Algorithm SORT_IRAN_BSP is outlined in Figure 3.

Compared to SORT_RAN_BSP, in SORT_IRAN_BSP local sorting is performed first on a set of n/p keys, not $(1 + 1/\omega_n)n/p$. Binary search of the input keys into the splitters can be simplified by merging the two in linear time or performing a binary search of the splitters into the input key set, a less expensive operation; the latter operation was implemented in the code of our experiments. Communication is simpler, as by the initial sorting and the binary-search operation of the splitter keys into the sorted local input keys, each processor is able to communicate a contiguous block of keys to every other processor. Because each processor receives sorted sequences a multi-way merge operation at the conclusion of the algorithm is required for the same asymptotic cost in terms of comparisons performed to the binary search operation of the keys into the splitter that has become unnecessary.

Computation time of SORT_IRAN_BSP is $n/p \lg(n/p) + (1 + 1/\omega_n)n \lg p/p + 2\omega_n^2 \lg n \lg^2 p + O(p \lg(n/p) + \omega_n^2 \lg n \lg p)$ and communication time is $(1 + 1/\omega_n)ng/p + g\omega_n^2 \lg n \lg^2 p + L \lg^2 p/2 + O(L \lg p + g\omega_n^2 \lg n \lg p + pg)$. The first two terms of computation time are due to local sorting and multi-way merging respectively,

```

begin SORT_IRAN_BSP ( $X$ )
1.  let  $s = 2\omega_n^2 \lg n$  ;
2.  for each processor  $\langle k \rangle$ ,  $k \in \{0, \dots, p-1\}$ , in parallel
3.    do SORT_SEQ( $X^{(k)}$ ) ;
4.    select uniformly at random a sample  $Y = \langle y_1, \dots, y_{sp-1} \rangle$  of  $sp-1$  keys ;
5.  let  $\bar{Y} = \text{Bitonic\_Sort}(Y)$ .
6.  for each processor  $\langle k \rangle$ ,  $k \in \{0, \dots, p-1\}$ , in parallel
7.    form set  $S = \langle s_1, \dots, s_{p-1} \rangle$  of  $p-1$  evenly spaced splitters
      that partition  $\bar{Y}$  into  $p$  evenly sized segments ;
8.    communicate the splitters to processor 0.
9.  BROADCAST( $S$ ) ;
10. for each processor  $\langle k \rangle$ ,  $k \in \{0, \dots, p-1\}$ , in parallel
11.   do partition  $X^{(k)}$  into  $p$  subsequences  $X_0^{(k)}, \dots, X_{p-1}^{(k)}$  as induced by the
       $p-1$  splitters of  $S$  ;
12.   for all  $i \in \{0, \dots, p-1\}$ 
13.     do communicate subsequence  $X_i^{(k)}$  to  $Z_k^{(i)}$  ;
14.   let  $\bar{X}^{(k)} = \text{MERGE\_SEQ}(\bigcup_i Z_i^{(k)})$  ;
15.   return  $\bar{X}^{(k)}$  ;
end SORT_IRAN_BSP

```

Figure 3: Procedure SORT_IRAN_BSP.

and the third term is due to parallel sample sorting. The first term of communication time is due to key routing, the second term is due to parallel sample sorting, and the third term reflects the number of synchronization operations required for parallel (Batcher-based) sample sorting. The terms hidden in the $O(\cdot)$ notation describe contributions of the remaining auxiliary operations. For example, in computation time the first term in $O(\cdot)$ reflects the cost of binary search of the splitters into the local keys, and the second term low order contributions of parallel (Batcher based) sample sorting. Similarly for communication time the first and seconds terms in $O(\cdot)$ reflect lower order contributions in parallel sample sorting and the third term the cost of splitter broadcasting. Therefore for $p^2 \leq n/(\omega_n \lg n)$ and $2p\omega_n^2 \lg p < n/2$, and $L \leq 2n/(p \lg^2 p)$, we conclude that $\pi = 1 + \lg p/(\omega_n \lg n) + 2p\omega_n^2 \lg^2 p/n + O(1/\omega_n \lg n + \omega_n^{3/2} \lg p/\sqrt{n \lg n})$ and $\mu = (1 + 1/\omega_n)g/\lg n + gp\omega_n^2 \lg^2 p/n + Lp \lg^2 p/(2n \lg n) + O(1/\lg p \lg n + g\omega_n^{3/2} \lg p/\sqrt{n \lg n} + g/(\omega_n \lg^2 n))$. Proposition 5.3 is then derived.

Proposition 5.3 *For any ω_n such that $2p\omega_n^2 \lg n < n/2$, $p^2 \leq n/(\omega_n \lg n)$ and $L \leq 2n/(p \lg^2 p)$, algorithm SORT_IRAN_BSP is such that $\pi = 1 + \lg p/(\omega_n \lg n) + 2p\omega_n^2 \lg^2 p/n + O(1/\omega_n \lg n + \omega_n^{3/2} \lg p/\sqrt{n \lg n})$ and $\mu = (1 + 1/\omega_n)g/\lg n + gp\omega_n^2 \lg^2 p/n + Lp \lg^2 p/(2n \lg n) + O(1/\lg p \lg n + g\omega_n^{3/2} \lg p/\sqrt{n \lg n} + g/(\omega_n \lg^2 n))$.*

6 Performance Evaluation

As the BSP model is not just an abstract architectural model, it may also be employed as a programming platform or, indeed, as a kind of a programming paradigm. The underlying concept in the BSP model is the notion of the superstep and the abstraction that non-local communication associated with a superstep takes place between supersteps as a global operation. Thus, a BSP program may be viewed as a succession of supersteps, with the required non-local communication occurring at the end of each superstep. Viewed

this way, the BSP model can be realized as a library of functions with architecture independent semantics for process creation, remote data access and bulk synchronization. The Oxford BSP Toolset, *BSPlib*, implements such a paradigm [43] and provides library support for BSP programming. The library functions are callable from standard imperative languages such as C and Fortran.

It should be borne in mind that such support can be made available by other non BSP-specific libraries (e.g. MPI) that support the simple communication and synchronization primitives required for programming under the BSP model. The effort required to learn the basics of *BSPlib* is minimal. One needs to understand the semantics of no more than 10-15 functions; half of these functions are related to process creation, destruction and identification. The implementations presented in this work were programmed in ANSI C. Only recompilation is required to run the same code on other platforms such as Silicon Graphics Origin 2000 and Power Challenge systems, or an IBM SP2 system. Our implementations are test for scalability and portability on a distributed memory system, a 128-processor Cray T3D. The manufacturer-supplied C compiler (`cc`) is used through the *BSPlib* front-end, and the source-code is compiled with the `-O3` compiler option set. Timing is obtained through the use of the real-time (wall-clock time) clock function `bsp_time` of *BSPlib* [37]. The timing results obtained are discussed later in this section. The depicted results in the following tables are in general, averages over at least four experiments. In the following discussion we shall require the BSP parameters of the machine configurations test. The CRAY T3D is thus reported to behave as a BSP machine with sets of parameters $(16, 130\mu\text{sec}, 0.21\mu\text{sec}/\text{int})$, $(32, 175\mu\text{sec}, 0.26\mu\text{sec}/\text{int})$, $(64, 364\mu\text{sec}, 0.28\mu\text{sec}/\text{int})$, $(128, 762\mu\text{sec}, 0.34\mu\text{sec}/\text{int})$, for the configuration used in the experiments (data type in communication is a 64-bit integer). Our implementation of quicksort, sorts 1024×1024 integer keys in about 3 seconds. This is equivalent to 7 comparisons per microsecond; expressing g in terms of basic computational operations (i.e. comparisons) a value of g of $0.21\mu\text{sec}/\text{int}$ becomes $0.21 \times 7 \approx 1.47$ comparisons/int. The BSP approach in modeling the performance and running time characteristics of parallel algorithms and programs is by modeling some abstract features of the underlying parallel platform as expressed through the parameters p , L and g . This parametrization is not as pure in terms of measuring hardware performance as say that of the LogP model. In fact it may be considered as more flexible, accommodating and forgiving. Parallel performance prediction has also attempted to model high-level parallel and sequential operations [17]. The problem with such approaches as also reported in [17] is that computational operation modeling can be affected by other platform characteristics (e.g. caching) thus making the study of scalability issues difficult. The minimalist approach of the BSP model may lead to fewer such problems, in general.

As the objectives of algorithm design on the BSP model are *scalability*, *portability* and *predictability of performance*, the algorithms we suggest may not lead to the best possible implementation on a specific platform. The sequential methods used may not be the best possible. One may also improve performance (sequential and parallel) by directly using specific features and interfaces of each machine (e.g. communication primitives). Our aim is to show that all three objectives can be realized without significant loss of performance in a general, architecture independent way. This is similar to the portability/reusability one obtains by programming in a high-level language as opposed to assembly.

6.1 Algorithm Implementation Features

In the implementation of the BSP algorithms, the following choices have been made.

- The implemented parallel algorithms are *comparison-based* sorting algorithms even though the

data type of the input keys is ANSI C `int` integers, and one of the two methods used for sequential sorting is an integer-specific method, radixsort. We use integer data keys for input as most other experimental studies also sort integer data and we would like to be able to compare our implementations to other ones. In addition, comparisons performed on integers are faster than those performed on strings of arbitrary length or non-trivial structures. This way we make sequential operations as fast as possible for the purpose of showing the parallel (communication) efficiency of our designs and implementations.

- Our algorithms *naturally handle duplicate keys* efficiently. The approach to handle duplicate keys is the one discussed in Section 5.1.1. We used it for both the deterministic and randomized algorithm implementations. As a result, the introduced algorithms are independent of input distribution. As it was discussed in Section 5.1.1, the memory overhead of handling duplicate keys is negligible, and may triple in the worst case the sample size as it attaches to each sample key an integer processor identifier and an integer array index; as sample size is $o(1)$ of problem size such overhead is tolerable. The overhead of duplicate handling in computation and communication time is in general asymptotically negligible thus affecting only lower order terms of the running time (a 3-6% deterioration in performance was observed in most of the experiments compared to test cases where duplicate key handling was intentionally switched off). The effect of duplicate key handling becomes non-negligible when sorting 1M integers (relatively few keys) on 128 (many) processors. Our approach is in contrast to the methods of [39, 40] that require twice as much communication to accommodate duplicate keys.
- Given the scarcity of experimental study results that can be used for comparison purposes and the fact that experimental sorting studies (e.g. [39, 40]) with integer input data use radixsort for sequential sorting, our implementations and experimental studies use two different algorithms for sequential sorting. One of the sequential sorting algorithms is a purely comparison-based algorithm, an author-written version of *quicksort*, and another one is an author-written integer specific version of *radixsort*. Radixsort was implemented purely for the purpose of comparing our implementations with other comparison-based parallel algorithms (such as the ones in [39, 40, 41]) that nevertheless use radixsort for sequential sorting of integers.
- Previous experimental results [31] describe generic implementations of our algorithms on the test and other platforms. A generic implementation is one that can generically sort any data type. This is achieved by including in the parameters of the parallel sorting function an additional argument, which is a function called `compare`, that compares two instances of the data type that is used as input for sorting. Standard C library function `qsort` is such an example of a generic sorting function. A function like `compare` returns -1, 0, or +1 depending on whether the former of its arguments is less, equal or greater than the latter one. The performance of a generic sorting algorithm that performs $O(n \lg n)$ comparisons to sort n keys is then bound by the $O(n \lg n)$ number of function calls to `compare`. As a consequence, a generic sorting function becomes 4-7 times slower than the corresponding non-generic one as it is also evident in the comparison of [39] (Table X) that compares a non-generic to a generic implementation. As all other experimental parallel sorting studies use non-generic implementations, we decided to follow their approach in this study. For a non-generic implementation however, one needs to create a new set of functions for each data type used. We therefore decided to implement

non-generic functions of the otherwise originally generic code that operate on integers.

- Total sample size over all the processors for the deterministic algorithm is $p^2 \lceil \omega_n \rceil$, where $\omega_n = \lg \lg n$. For the randomized algorithm it is $2p\omega_n^2 \lg n$, where $\omega_n^2 = \lg n$.

6.2 Algorithm implementations

The following BSP sorting algorithms have been implemented on top of *BSPlib*.

- (1) Two variants of the one-optimal deterministic algorithm `SORT_DET_BSP` that both operate on ANSI C `int` data types, with duplicate key handling performed according to the method described in Section 5.1.1. One uses for sequential sorting, quicksort (an author written implementation) and is called `[DSQ]`, and the other uses radixsort and called `[DSR]`.
- (2) Two variants of the one-optimal randomized algorithm `SORT_IRAN_BSP`, with duplicate key handling performed according to the method described in Section 5.1.1, called, similarly to the previous case, `[RSQ]` and `[RSR]`.
- (3) A version of Batcher’s bitonic sort [5] has been implemented and is called `[BSI]`. `[BSI]` is used for parallel sample sorting only. As its performance is worse than that of any of the other four implementations in all but very small problem and processor sizes (for such cases, Batcher’s algorithm is faster because of its low overhead) we do not compare its performance to our other implementations.

Remark 1. The implementations can handle duplicate keys as this was previously explained.

Remark 2. The implementations are *portable* and *reusable* enough to allow any sequential sorting or merging algorithm to be used as the underlying sequential method.

6.3 Sorting Benchmarks

We have test our implementations on a variety of input sets. We tried to conform to previously published data sets [39, 40, 41] by including them in this experimental study. For a discussion of the practicality and suitability of this set of sorting benchmarks we refer to [41, 40, 39]. The sorting benchmarks are briefly defined and `INT_MAX` is the maximum integer value plus one accommodated in a 32-bit signed arithmetic data type (e.g., 2^{31}). The definitions below follow those appearing in [41], except for the worst regular input set that follows the definition in [39, 40]. The format of the tables reporting the results is similar to the ones of [39, 40, 41]. In addition, we decided not to test our implementation on two additional data sets of [39, 40], i.e. `[Z]` and `[RD]` for one good reason. Due to the way our duplicate-key handling method works, some preliminary results for these two distributions were in no case worse than that of `[U]` (in fact, they were better) and similar to those of `[DD]` or `[WR]`. These observations agree with the results of [39] where `[Z]` and `[RD]` give results similar to `[DD]` and no worse than those of `[U]`, and of [40] where the results of `[Z]` and `[RD]` are similar to `[DD]` and no worse than those of `[U]`.

- (1) Uniform `[U]`, the input is uniformly and at random distributed, and is generated by calling a pseudo random number generator, the C standard library function `random()`, which returns a long (integer) in the range $[0, \dots, 2^{31} - 1]$ and processor’s i seed is $21 + 1001 \cdot i$.

- (2) Gaussian [G], the input follows a Gaussian distribution, and is approximated by adding the results of four calls to `random()` and dividing the sum by four.
- (3) Bucket Sorted [B], the input (per processor) is split into p buckets, each of size n/p^2 , so that the i -th such bucket, $0 \leq i \leq p - 1$, consists of numbers uniformly and at random distributed in the range $[i \text{INT_MAX}/p, \dots, (i + 1) \text{INT_MAX}/p - 1]$.
- (4) g -Group [g-G], the processors are first divided into p/g groups each consisting of g processors, and within group j each processors splits its input into g buckets so that the i -th such bucket consists of numbers uniformly and at random distributed in the range $[((jg + p/2 + i) \bmod p) \text{INT_MAX}/p, \dots, ((jg + p/2 + i + 1) \bmod p) \text{INT_MAX}/p - 1]$.
- (5) Staggered [S], the input of processor $i < p/2$ is uniformly and at random distributed in the range $[(2i + 1) \text{INT_MAX}/p, \dots, (2i + 2) \text{INT_MAX}/p - 1]$, and of processor $i \geq p/2$ in the range $[(i - p/2) \text{INT_MAX}/p, \dots, (i - p/2 + 1) \text{INT_MAX}/p - 1]$.
- (6) Deterministic Duplicates [DD], the $n/2^i$ input keys of the first $p/2^i$ consecutive processors are all set to $\lg(n/p^{i-1})$, and so forth. In processor $p - 1$, $n/(p2^i)$ keys, starting from the lower indexed and proceeding to higher indexed, are set to $\lg n/(p2^{i-1})$ and so forth.
- (7) Worst Regular [WR], as described in [39].

6.4 Experimental Results

In this section, we report on the performance and relative merits of the implementations of the algorithms of Section 5. The tables below summarize some of the experimental results we had obtained. Timing figures are in general truncated to three or fewer decimal digits.

Table 1 shows timing results for algorithm `SORT_IRAN_BSP` on 64 processors of a Cray T3D. Results for both [RSR] and [RSQ] are presented for all seven benchmark input sets. Size refers to the total size n of the input over all 64 processors.

It is possible to compare our implementation of `SORT_IRAN_BSP` to the one described and implemented in [41] (Table I, page 215). For problem size 1M, ours is slower by about 3-10%. For the other common problem sizes (4M, 16M, 64M) our implementation is faster by about 3-8%. It is worth noting that for small problem sizes (e.g. 1M) the use of quicksort for sequential sorting improves the parallel performance of our implementation by 1-2%.

Compared to the implementation of the randomized sorting algorithm introduced in [40], our implementation is slower by 3-15% for problem size 1M, 3-5% for problem size 4M, and about 1-2% for the remaining problem sizes. The slowness of our implementation is mainly attributable to sequential merging which takes 33-39% of the total execution time of any one experiment as opposed to only 25% for [40]. Sequential operations (sorting and merging related) take together 90% of execution time in our implementation and 80% in the implementation of [40]. Although our implementation is more communication efficient, mainly because it uses only one round of data communication, this advantage is wasted by the apparent inefficiency of a sequential operation (merging).

The experimental results obtained for all seven benchmark sets for algorithm `SORT_DET_BSP` are depicted in Table 2. A comparison of our implementation to the one in [41] (Table IV, page 218) indicates that ours is faster by as much as 13-35% for problem size 1M, 15-25% for problem size 4M, 12-15% for 16M, and 7-15% for 64M. This is possibly attributable to the fact that the algorithm in [41] is a direct

Running Time of SORT_IRAN_BSP on 64 procs														
Size	[RSR]							[RSQ]						
	[U]	[G]	[2-G]	[B]	[S]	[DD]	[WR]	[U]	[G]	[B]	[2-G]	[S]	[DD]	[WR]
1M	0.079	0.077	0.073	0.079	0.062	0.053	0.079	0.076	0.078	0.072	0.068	0.062	0.050	0.067
4M	0.269	0.269	0.254	0.270	0.211	0.171	0.271	0.283	0.280	0.258	0.239	0.234	0.176	0.241
8M	0.526	0.526	0.494	0.527	0.408	0.329	0.528	0.559	0.563	0.522	0.484	0.462	0.356	0.486
16M	1.03	1.03	0.987	1.04	0.802	0.643	1.04	1.15	1.16	1.10	1.00	0.960	0.735	1.00
32M	2.06	2.07	1.92	2.06	1.60	1.27	2.06	2.39	2.37	2.24	2.05	2.00	1.54	2.06
64M	4.09	4.09	3.90	4.09	3.16	2.50	4.11	4.88	4.88	4.65	4.29	4.16	3.17	4.31

Table 1: Execution time of SORT_IRAN_BSP with $p = 64$ ($1M = 1024 \times 1024$).

Running Time of SORT_DET_BSP on 64 procs														
Size	[DSR]							[DSQ]						
	[U]	[G]	[2-G]	[B]	[S]	[DD]	[WR]	[U]	[G]	[B]	[2-G]	[S]	[DD]	[WR]
1M	0.091	0.091	0.084	0.089	0.073	0.063	0.089	0.088	0.088	0.083	0.079	0.072	0.060	0.079
4M	0.281	0.280	0.265	0.279	0.218	0.177	0.280	0.294	0.291	0.269	0.249	0.242	0.183	0.250
8M	0.532	0.530	0.507	0.529	0.415	0.330	0.529	0.566	0.566	0.535	0.487	0.470	0.360	0.489
16M	1.03	1.03	0.985	1.03	0.803	0.633	1.03	1.15	1.16	1.09	0.985	0.963	0.727	0.990
32M	2.06	2.06	1.92	2.07	1.58	1.24	2.07	2.38	2.37	2.26	2.06	1.98	1.50	2.07
64M	4.09	4.09	3.88	4.08	3.12	2.44	4.08	4.94	4.86	4.63	4.27	4.18	3.14	4.27

Table 2: Execution time of SORT_DET_BSP with $p = 64$ ($1M = 1024 \times 1024$).

variant of the one in [61]. Our implementation on the other hand, although it is also based on ideas of [61], it naturally handles duplicate keys (without any significant increase of communication time), implements deterministic oversampling and determines sample size on the basis of maximum key imbalance at the completion of sorting. Its design and analysis is solely based on the BSP model and seems to perform better in practice exhibiting performance that is within the predictions of the theoretical analysis.

The algorithm of [41] was further refined into an oversampling-based algorithm that takes into account duplicate keys and is described in [39]. This latter algorithm also handles duplicate keys by performing twice as much communication. Comparing the performance of that latter algorithm (Table III of [39]) to our results as obtained in Table 2 we observe that for problem size 1M our implementation is faster by as much as 10-20%, and for problem sizes, 4M, 16M and 64M our algorithm is faster by 8-26%, 3-10% and 2-3%. In our implementation sorting requires one communication round instead of two in [39]. Even though our sequential methods are slower (as exhibited in Table 4) than the ones implemented in [39] (Table IX), the extra communication round of the algorithm in [39] neutralizes such an advantage even though the cost of a communication round is no more than 5-6% of the overall time of the algorithm as opposed to sequential sorting and merging that account for 45-60% and 30-40% of the running time respectively.

Input Set	[RSR]					[RSQ]				
	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
[U]	3.16	1.74	0.956	0.526	0.300 (65%)	3.90	2.00	1.07	0.559	0.310 (78%)
[WR]	3.16	1.74	0.956	0.527	0.306 (64%)	3.64	1.82	0.938	0.486	0.272 (83%)
Input Set	[DSR]					[DSQ]				
	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
[U]	3.18	1.72	0.947	0.532	0.374 (53%)	3.92	1.98	1.07	0.566	0.386 (63%)
[WR]	3.18	1.73	0.945	0.530	0.372 (53%)	3.65	1.82	0.930	0.489	0.337 (67%)

Table 3: Execution time of SORT_IRAN_BSP and SORT_DET_BSP on an input of size $8 \times 1024 \times 1024$. For $p = 128$ efficiencies are also shown.

Table 3 shows the scalability of our algorithm implementations on inputs [U] and [WR]. We used these two input distributions as they have also been used in the experimental work of [39, 40, 41]; in those studies,

for a deterministic sorting algorithm, [WR] is used exclusively [39, 41], whereas for a randomized sorting algorithm both [WR] [41] and [U] [40] are used. A scalability comparison of all these algorithms is depicted in Table 9 later in this work. With respect to Table 3 it is worth noting that for relatively small ratios n/p (e.g. when $p = 128$ and for moderate problem sizes), sequential sorting of integers using quicksort may yield better performance than using radixsort, as expected. As the variants of our algorithms that use quicksort for sequential sorting are more CPU intensive, efficiencies are higher for such cases than those that use radixsort. For $p = 128$, [RSR] has the same efficiency 64-65% independent of the input data distribution; for [RSQ] this varies between 78% to 83%. The efficiencies of the deterministic algorithm are lower however; they are 53% and vary between 63% - 67%, i.e. they are between 12-15% lower than those of the randomized algorithm.

This is due to the fact that for a fixed size sample, random sampling yields more balanced set of keys in the routing and multi-way merging phases of the sorting algorithm. We note that for the deterministic algorithm we chose $\omega_n = \lg \lg n$ and for the randomized algorithm $\omega_n^2 = \lg n$. In all runs of our experiments (of either the deterministic or the randomized algorithm) maximum set imbalance was kept below 15% well within the approximately 20% obtained from the theoretical analysis for the deterministic ($1/\lceil \lg \lg n \rceil \times 100\%$) and randomized algorithms ($1/\sqrt{\lg n} \times 100\%$).

For problem size $n = 2^{23} = 8M$ as used in Table 3 and for $p = 128$, the conditions on n , p and L in Proposition 5.1 and 5.3 are hardly satisfied. Substituting for n , p , L and g in the expressions for π and μ in Proposition 5.1 and ignoring the low order terms (enclosed in $O(\cdot)$) we derive a theoretical bound on efficiency of at least 66% for [DSQ]. The observed efficiency was 63% and 67% for [U] and [WR] respectively. We note that the contribution to the theoretical estimation of running time of handling duplicate keys was ignored (such contributions would have been absorbed in the $O(\cdot)$ term which is anyway ignored in the calculation of the theoretical efficiency). Had we disabled the code for handling duplicate keys, for $p = 128$ running time for [DSQ] on input [U] would be 0.348 (i.e. efficiency 70% instead of 63%) and that for [DSR] on input [U] would be 0.336 (i.e. efficiency 58% instead of 53%). For the randomized algorithm the theoretical prediction of at least 66% was also satisfied in practice (observed efficiency was 78-82%). We note that the theoretical analysis applies only to the case that sequential sorting is performed by a comparison and exchange algorithm only. Although the execution time of radixsort is independent of the input distribution (if we ignore caching effects), that of quicksort is not.

Procs	Time per phase						Percentage (%) of total time for each phase					
	8M			32M			8M			32M		
	32	64	128	32	64	128	32	64	128	32	64	128
Ph 1	0.000	0.001	0.002	0.000	0.001	0.002	0.07	0.23	0.73	0.02	0.06	0.20
Ph 2	0.560	0.277	0.137	2.220	1.115	0.557	57.76	52.64	45.48	58.62	54.09	49.49
Ph 3	0.011	0.011	0.017	0.011	0.013	0.020	1.13	2.11	5.61	0.29	0.63	1.77
Ph 4	0.004	0.003	0.006	0.004	0.003	0.006	0.41	0.57	1.98	0.11	0.15	0.53
Ph 5	0.066	0.036	0.021	0.259	0.144	0.080	6.80	6.82	6.93	6.83	6.98	7.10
Ph 6	0.324	0.198	0.118	1.288	0.786	0.461	33.40	37.57	38.94	33.98	38.10	40.91
Ph 7	0.004	0.000	0.001	0.006	0.000	0.000	0.41	0.06	0.33	0.16	0.00	0.00
Total	0.970	0.527	0.303	3.791	2.063	1.127	100	100	100	100	100	100

Table 4: Scalability of various phases of [RSR] on input [U]. Ph1=Init, Ph2=SeqSort, Ph3=Sampling, Ph4=Prefix, Ph5=Routing, Ph6=Merging, Ph7=Termination.

Tables 4 and 5 show the distribution of execution time for sorting 8M and 32M integers by the randomized algorithm when input is [U]. Percentages of total running time for the execution time of each phase are also shown. With reference to Figure 3 and Tables 4/5, Ph1 corresponds to the part of the code before

Procs	Time per phase						Percentage (%) of total time for each phase					
	8M			32M			8M			32M		
	32	64	128	32	64	128	32	64	128	32	64	128
Ph 1	0.000	0.001	0.002	0.000	0.001	0.002	0.06	0.25	0.76	0.02	0.05	0.18
Ph 2	0.675	0.311	0.150	3.007	1.432	0.675	61.56	55.48	47.81	65.70	60.00	54.37
Ph 3	0.010	0.010	0.017	0.011	0.013	0.019	0.91	1.78	5.40	0.24	0.54	1.53
Ph 4	0.014	0.003	0.005	0.005	0.003	0.006	1.28	0.53	1.59	0.11	0.13	0.48
Ph 5	0.071	0.038	0.021	0.262	0.142	0.078	6.47	6.77	6.67	5.72	5.95	6.28
Ph 6	0.323	0.197	0.119	1.288	0.796	0.462	29.44	35.13	37.78	28.14	33.33	37.17
Ph 7	0.003	0.000	0.000	0.003	0.000	0.000	0.27	0.05	0.00	0.07	0.00	0.00
Total	1.097	0.561	0.315	4.577	2.388	1.243	100	100	100	100	100	100

Table 5: Scalability of various phases of [RSQ] on input [U]. Ph1=Init, Ph2=SeqSort, Ph3=Sampling, Ph4=Prefix, Ph5=Routing, Ph6=Merging, Ph7=Termination.

line 2, Ph2 (sequential sorting) to the execution of lines 2-3, Ph3 to lines 4-9, Ph4 to lines 9-12, Ph5 (key routing) to lines 12-13, and Ph6 (sequential merging) to lines 14-15 of the code for SORT_IRAN_BSP. From both tables, the sequential portion of the code contributes at least 85-90% of the running time. Efficient sequential implementations can thus significantly affect the execution time of our implementations, as it was remarked when a comparison of our implementation to other ones was made.

It is worth observing that routing time of Ph5 scales satisfactorily with the number of processors, an indication that the randomized algorithm is quite scalable. In the implementation, $\omega_n = \sqrt{\lg n}$, and therefore in Ph6 each processor is expected to send or receive no more than $(1 + 1/\omega_n)n/p$ keys. The observed imbalance on any of the experiments reported in Tables 4 and 5 was kept below 15% which is within the 20% imbalance implied by the theoretical calculations (e.g. $1/\sqrt{\lg 2^{23}} \approx 0.208$). The communication time of Ph6 gives an observed value for g of 0.23 – 0.25, 0.24 – 0.28 and 0.28 – 0.32 for $p = 32$, $p = 64$ and $p = 128$ respectively (assuming that any processor sent or received at most $1.15n/p$ keys) and these values are consistent with the measured ones (in other experiments [35]) of 0.26, 0.28 and 0.34 $\mu\text{sec/int}$ respectively reported in the beginning of this section.

Procs	Time per phase						Percent(%) of total time per phase					
	8M			32M			8M			32M		
	32	64	128	32	64	128	32	64	128	32	64	128
Ph 1	0.000	0.001	0.002	0.000	0.001	0.002	0.07	0.23	0.59	0.02	0.06	0.18
Ph 2	0.560	0.277	0.139	2.222	1.115	0.558	57.94	53.42	41.12	58.64	54.30	49.73
Ph 3	0.005	0.008	0.018	0.006	0.008	0.018	0.52	1.54	5.33	0.16	0.39	1.60
Ph 4	0.004	0.003	0.005	0.005	0.003	0.005	0.41	0.58	1.48	0.13	0.15	0.45
Ph 5	0.073	0.038	0.022	0.260	0.145	0.081	7.55	7.31	6.51	6.86	7.06	7.22
Ph 6	0.315	0.192	0.152	1.294	0.781	0.458	32.57	36.92	44.97	34.14	38.00	40.82
Ph 7	0.009	0.000	0.000	0.002	0.001	0.000	0.93	0.00	0.00	0.05	0.05	0.00
Total	0.967	0.520	0.338	3.79	2.055	1.122	100	100	100	100	100	100

Table 6: Scalability of various phases of [DSR] on input [U]. Ph1=Init, Ph2=SeqSort, Ph3=Sampling, Ph4=Prefix, Ph5=Routing, Ph6=Merging, Ph7=Termination.

Tables 6 and 7 show the distribution of execution time for sorting 8M and 32M integers by the deterministic algorithm when input is [U]. Percentages of total running time for the execution time of each phase are also shown. With reference to Figure 1, Ph1 corresponds to the part of the code before line 2, Ph2 (sequential sorting) to the execution of lines 2-3, Ph3 to lines 4-7, Ph4 to lines 8-9, Ph5 (key routing) to lines 10-11, and Ph6 (sequential merging) to lines 12-13 of the code for SORT_DET_BSP. Communication performance is scalable though slightly worse than those of the randomized algorithm as random

Procs	Time per phase						Percent(%) of total time per phase					
	8M			32M			8M			32M		
	32	64	128	32	64	128	32	64	128	32	64	128
Ph 1	0.000	0.001	0.002	0.000	0.001	0.002	0.06	0.22	0.57	0.02	0.05	0.16
Ph 2	0.675	0.310	0.151	3.005	1.433	0.676	62.76	56.2	43.14	65.46	60.32	54.6
Ph 3	0.006	0.008	0.018	0.006	0.008	0.017	0.56	1.45	5.14	0.13	0.34	1.37
Ph 4	0.004	0.003	0.005	0.005	0.003	0.005	0.37	0.54	1.43	0.11	0.13	0.4
Ph 5	0.071	0.037	0.022	0.268	0.150	0.077	6.60	6.69	6.29	5.84	6.31	6.22
Ph 6	0.316	0.192	0.151	1.303	0.780	0.460	29.37	34.72	43.14	28.38	32.81	37.16
Ph 7	0.003	0.001	0.001	0.003	0.001	0.001	0.28	0.18	0.29	0.07	0.04	0.08
Total	1.076	0.553	0.350	4.591	2.377	1.238	100	100	100	100	100	100

Table 7: Scalability of various phases of [DSQ] on input [U]. Ph1=Init, Ph2=SeqSort, Ph3=Sampling, Ph4=Prefix, Ph5=Routing, Ph6=Merging, Ph7=Termination.

	Time per common phase						Percent(%) of total time per phase					
	[DSR]on [U]			[39] on [WR]			[DSR]on [U]			[39] on [WR]		
	32	64	128	32	64	128	32	64	128	32	64	128
Ph 2	0.560	0.277	0.139	0.591	0.299	0.151	57.94	53.42	41.12	60.60	50.41	30.52
Ph R	-	-	-	0.045	0.029	0.019	-	-	-	4.62	4.88	3.94
Ph 5	0.073	0.038	0.022	0.050	0.039	0.076	7.55	7.31	6.51	5.13	6.65	15.35
Ph 6	0.315	0.192	0.152	0.215	0.152	0.107	32.57	36.92	43.97	22.06	25.65	21.64
Total	0.967	0.52	0.338	0.976	0.593	0.496	100	100	100	100	100	100

Table 8: Scalability comparison of [DSR] and [39]. Ph2=SeqSort, PhR,Ph5=Routing, Ph6=Merging.

oversampling causes more balanced communication.

From Tables 4 and 6 and Tables 5 and 7, the sequential portion of the code contributes at least 86-93% of the running time. As the two algorithms share most operations except sampling, the performance of the two algorithms in the first few phases is similar. The quality of sampling affects Phases 5 and 6. This is evident for example from the figures in the four tables for $n = 8M$ and $p = 128$. The randomized algorithm is about 0.033 seconds faster than the deterministic algorithm and this is solely due to key-imbalance and the better balancing properties of the randomized algorithm whose oversampling parameter can vary more widely than that of the deterministic algorithm.

Table 8 gives comparative timing results for each one of the major phases of [DSR] and the algorithm in [39] on inputs [U] and [WR] respectively. As for our implementation, the running time of [DSR] on either input [U] or [WR] are almost identical, the phase by phase timing of our implementation does not change when input is [WR] rather than [U]. From Table 8 we conclude that even if our sequential methods are slower for multi-way merging than those of [39] the balanced single round of communication in [DSR] is more scalable than the corresponding round of [39] (listed under Ph 5 in Table 8), even though the portable communication library of our experiments may be less efficient than the one of [39, 40, 41]. This may be attributable to sampling choices or the way duplicate keys are handled in [39].

In Table 9 the first four rows compare the performance of [RSR] to the ones in [40, 41] for problem size $n = 8M$. Performance of the compared algorithms is similar for up to $p = 64$ processors except for $p = 128$ where our implementation is slightly worse by about 5% attributable mainly to some inefficiency in sequential merging. The following three rows compare the performance of [DSR] to the deterministic regular sampling algorithms in [39, 41]. [DSR] is more efficient than the algorithm in [41] through all processor ranges by about 15-30%. Compared to the implementation in [39] the two exhibit about the same performance for up to $p = 32$ (the disadvantage of our sequential merging is counterbalanced by the two rounds of communication of [40]), and for $p = 64$ and $p = 128$ [DSR] is better by 10% and 30%

Algorithm	Input	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
[RSR]	[U]	3.16	1.74	0.956	0.526	0.300
[40]	[U]	3.32	1.77	0.952	0.513	0.284
[RSR]	[WR]	3.16	1.74	0.956	0.527	0.306
[41]	[WR]	3.21	1.74	0.966	0.540	0.294
[DSR]	[WR]	3.18	1.73	0.945	0.530	0.372
[41]	[WR]	4.07	2.11	1.150	0.711	-
[39]	[WR]	3.23	1.73	0.976	0.594	0.496
[DSQ]	[WR]	3.65	1.82	0.930	0.489	0.337
[RSQ]	[WR]	3.64	1.82	0.938	0.486	0.272
[DSQ]	[U]	3.92	1.98	1.066	0.566	0.386
[RSQ]	[U]	3.90	2.00	1.070	0.559	0.310
[DSR]	[U]	3.18	1.72	0.947	0.532	0.374

Table 9: Comparison of our results with other algorithm implementations.

respectively due to more scalable communication and the single communication round of [DSR].

	[DSR]					[DSQ]				
	8	16	32	64	128	8	16	32	64	128
1M	0.395	0.222	0.128	0.077	0.133	0.413	0.222	0.127	0.075	0.135
4M	1.574	0.869	0.480	0.280	0.270	1.807	0.985	0.514	0.294	0.280
8M	3.263	1.728	0.947	0.532	0.339	3.967	1.988	1.065	0.565	0.352
	[RSR]					[RSQ]				
	8	16	32	64	128	8	16	32	64	128
1M	0.399	0.223	0.126	0.079	0.060	0.416	0.223	0.124	0.076	0.057
4M	1.593	0.877	0.484	0.270	0.165	1.826	0.992	0.516	0.284	0.164
8M	3.161	1.746	0.957	0.527	0.302	3.915	2.00	1.074	0.558	0.314

Table 10: Scalability of [DSR], [RSR] and [DSQ],[RSQ] on same input [U].

Table 10 depicts the scalability of the four sorting variants introduced in this work with increasing processor sizes for three moderate inputs sizes. As mentioned earlier for $p = 128$ and $n = 8M$, the asymptotic claims for SORT_IRAN_BSP and SORT_DET_BSP hardly hold, and this is also evidenced by the deteriorating performance of the implementations for $n = 4M$ and $n = 1M$. The slowdown for $n = 1M$ is also attributable to the effect of the extra code for duplicate handling; disabling of this piece of code results in the elimination of the slowdown observed for $n = 1M$; the obtained speedup is however minimal. We note that the experiments reported in Table 10 constitute a separate set of experiments in addition to those performed for the creation of Table 9, Table 1, and Table 2; this explains slight differences in running times which for almost all cases affect the third decimal digit of various reported timing entries.

Another work that includes experimental results on deterministic sorting appears in [44]. Table 11 below compares [DSQ] to a direct implementation of the regular sampling algorithm of [61] reported in [44]. The implementation of [44] is similar to the deterministic algorithm of [41] and the performance of the two algorithms is similar and significantly worse than the more refined algorithm SORT_DET_BSP ([DSQ]) or the one in [39]. In addition, the algorithm in [44] as well as the algorithm in [41] can not handle duplicate keys.

Algorithm	Input	n	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
[DSQ]	[U]	1024×1024	0.413	0.222	0.127	0.075	0.135
[44]	[U]	1000000	0.462	0.240	0.137	0.117	-

Table 11: Comparison of [DSQ] on [U] with $n = 1024 \times 1024$ keys to [44] on [U] with $n = 1,000,000$ keys.

7 Conclusion

In this work we have introduced deterministic and randomized algorithms for internal memory sorting that were designed and their performance analyzed in an architecture independent way under the BSP model. The deterministic algorithm [28] is an improvement of the regular sampling algorithm of [61] and uses the technique of deterministic oversampling (used for randomized sorting in [59]) and in the general case, its theoretical performance is fully analyzed in [28]. Some other of its advantages are its parallel sample-sort that allows p to be much close to n than the algorithm in [61]. From our experimental observations it seems that parallel sample-sort is effective even for small problem sizes, even if other parallel sorting algorithms and implementations do not use it. In addition, the introduced in this work transparent and efficient duplicate handling method of Section 5.1.1 allows the algorithm to handle duplicate keys without doubling the computation load or the communication time that other approaches seem to require [39, 40, 41]. The randomized algorithm although oversample-sort based works differently from other sample-sort based approaches and even improves upon the performance of the first randomized BSP sorting algorithm of [21]; it achieves this by using ideas derived from the deterministic algorithm. It also handles duplicate keys with insignificant degradation in performance (computation or communication). Given the insistence under the BSP model of one-optimality and the accurate accounting of key imbalance during sorting, it is easy to fine-tune the work-load balance and communication time by adjusting the oversampling parameter and sample size of any of the two algorithms. The experimental results presented in this work verify the theoretical claims on the efficiency of the introduced algorithms. We have compared our implementations to other parallel sorting implementations on the same or similar platforms and realized that in terms of parallel communication efficiency our algorithms seem to outperform other implementations and overall exhibit comparable if not better performance (e.g. deterministic algorithm) even though the sequential algorithms used in our implementations seem, in many instances, to be slower than those of other implementations. This suggests that one can probably improve overall performance of our implementations if sequential methods are optimized given the fact that in our communication efficient designs sequential code takes 80-90% of execution time for the problem instances examined. As far as communication is concerned there is probably no room for significant improvement unless one tries to optimize the single communication round by utilizing platform specific communication primitives at the expense of portability.

The experimental portion of this work was performed while both authors were with the Oxford University Computing Laboratory, The University of Oxford, Oxford, UK. The work of the first author there was supported in part by EPSRC under grant GR/K16999 and that of the second author was supported in part by a Bodossaki Foundation Graduate Scholarship. The support of the Edinburgh Parallel Computing Centre in granting the first author access to a Cray T3D computer is gratefully acknowledged. The work of the first author was subsequently supported in part by NJIT SBR grant 421350 and NSF grant NSF-9977508.

References

- [1] M. Adler, J. W. Byers, and R. M. Karp. Parallel sorting with limited bandwidth. In *Proceedings of the 7-th ACM Symposium on Parallel Algorithms and Architectures*, pp. 129-136, ACM Press 1995.
- [2] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(\log n)$ sorting network. In *Proceedings of the 15-th Annual ACM Symposium on Theory of Computing*, pp. 1-9, 1983.
- [3] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ steps. *Combinatorica*, 3:1-19, Springer-Verlag, 1983.
- [4] A. G. Alexandrakis, A. V. Gerbessiotis and C. J. Siniolakis. "Portable and Scalable Algorithm Design on the IBM SP2: The Bulk-Synchronous Parallel Paradigm". In Proceedings of the SUP'EUR '96 Conference, Krakow, Poland, September 1996.
- [5] K. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, pp. 307-314, 1968.
- [6] G. Baudet and D. Stevenson. Optimal sorting algorithms for parallel computers. *IEEE Transactions on Computers*, C-27(1):84-87, January 1978.
- [7] A. Baumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: c -optimal multisearch for an extension of the BSP model. In *Proceedings of the Annual European Symposium on Algorithms*, 1995.
- [8] A. Baumker, and W. Dittrich. Fully Dynamic Search Trees for an Extension of the BSP Model. In *8th Annual ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy, ACM Press 1996.
- [9] A. Baumker, W. Dittrich, F. Meyer auf der Heide and I. Rieping. Realistic parallel algorithms: Priority queue operations and selection for the BSP* model. In *Proceedings of EUROPAR'96*, LNCS volume 1124, August 1996.
- [10] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP vs. LogP. In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, pp. 25-32, 1996.
- [11] R. H. Bisseling and W. F. McColl. Scientific computing on bulk-Synchronous Parallel architectures. Preprint 836, Department of Mathematics, University of Utrecht, December 1993.
- [12] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, pp. 3-16, 1991, ACM Press.
- [13] B. Bollobás. *Random Graphs*. Academic Press, New York, 1984.
- [14] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770-785, 1988.
- [15] D. E. Culler and R. Karp and D. Patterson and A. Sahay and K. E. Schauser and E. Santos and R. Subramonian and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, May 1993.

- [16] R. Cypher and J. Sanz. Cubesort: A parallel algorithm for sorting n -data items with s -sorters. *Journal of Algorithms*, 13:211-234, 1992.
- [17] A. C. Dusseau. Modeling parallel sorts with LogP on the CM-5. Technical Report UCB/CSD-94-829, Computer Science Division, University of California at Berkely, 1994.
- [18] M. H. van Emden. Increasing the efficiency of quicksort. *Comm. of the ACM*, 13:9:563-567, 1970.
- [19] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10-th Annual ACM Symposium on Theory of Computing*, pp. 114-118, 1978.
- [20] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM*, 17(3):496-507, July 1970.
- [21] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251-267, 1994.
- [22] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of the 8-th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 223-232, Padua, Italy, June 1996.
- [23] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications in computational geometry. In *Proceedings of EURO-PAR'96*, Lyon, France, Lecture Notes in Computer Science, Volume 1124, Springer-Verlag, August 1996.
- [24] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications. Technical Report PRG-TR-13-96, Computing Laboratory, Oxford University, June 1996.
- [25] A. V. Gerbessiotis and C. J. Siniolakis. Concurrent heaps on the BSP model. Technical Report PRG-TR-14-96, Computing Laboratory, Oxford University, June 1996.
- [26] A. V. Gerbessiotis and C. J. Siniolakis. Selection on the Bulk-Synchronous Parallel model with applications to priority queues. In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, Sunnyvale, California, USA, August 1996.
- [27] A. V. Gerbessiotis and C. J. Siniolakis. Efficient Deterministic Sorting on the BSP Model. Technical Report PRG-TR-19-96, Oxford University Computing Laboratory, October 1996.
- [28] A. V. Gerbessiotis and C. J. Siniolakis. Efficient deterministic sorting on the BSP model. *Parallel Processing Letters*, Vol 9 No 1 (1999), pp 69-79, World Scientific Publishing Company.
- [29] A. V. Gerbessiotis and C. J. Siniolakis. Primitive Operations on the BSP Model. Technical Report PRG-TR-23-96, Oxford University Computing Laboratory, October 1996.
- [30] A. V. Gerbessiotis and C. J. Siniolakis. A Randomized Sorting Algorithm on the BSP model. In *Proceedings of the International Parallel Processing Symposium*, Geneva, Switzerland, IEEE Press, 1997.

- [31] A. V. Gerbessiotis and C. J. Siniolakis. An Experimental Study of BSP Sorting Algorithms. In *Proceedings of 6th EuroMicro Workshop on Parallel and Distributed Processing*, Madrid, Spain, January, IEEE Computer Society Press, 1998.
- [32] A. V. Gerbessiotis, D. S. Lecomber, C. J. Siniolakis and K. R. Sujithan. PRAM Programming: Theory vs. Practice. In *Proceedings of 6th EuroMicro Workshop on Parallel and Distributed Processing*, pp. 164-170, Madrid, Spain, January, 1998, IEEE Computer Society Press.
- [33] A. V. Gerbessiotis and C. J. Siniolakis. Ordered h -Level Graphs on the BSP Model. *Journal of Parallel and Distributed Computing*, 49:98-110, Academic Press, 1998.
- [34] A. V. Gerbessiotis. Practical considerations of parallel simulations and architecture independent parallel algorithm design. In *Journal of Parallel and Distributed Computing*, 53:1-25, Academic Press, 1998.
- [35] A. V. Gerbessiotis, and F. Petrini. Network Performance Assessment under the BSP Model, In *International Workshop on Constructive Methods for Parallel Programming*, June 1998, Gothenborg, Sweden.
- [36] M. T. Goodrich. Communication-Efficient Parallel Sorting. In *Proceedings of the 26th ACM Symposium on the Theory of Computing*, ACM Press, 1996.
- [37] M.W. Goudreau, J.M.D. Hill, K. Lang, W.F. McColl, S.D. Rao, D.C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for a BSP Worldwide standard. BSP Worldwide, <http://www.bsp-worldwide.org/>, April 1996.
- [38] M. Goudreau, K. Lang, S. Rao and T. Tsantilas. The Green BSP Library. Tech. Rep. CR-TR-95-11, University of Central Florida, 1995.
- [39] D. R. Helman, J. JaJa, and D. A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. Technical Report UMIACS-TR-96-54/CS-TR-3670, The University of Maryland Institute for Advanced Computer Studies, August 1996.
- [40] D. R. Helman, D. A. Bader, and J. JaJa. A randomized parallel sorting algorithm with an experimental study. Technical Report UMIACS-TR-96-53/CS-TR-3669, The University of Maryland Institute for Advanced Computer Studies, August 1996. Also in *J. Parallel and Distributed Comput.*, 52(1):1-23, 1998.
- [41] D. R. Helman, D. A. Bader, and J. JaJa. Parallel algorithms for personalized communication and sorting with an experimental study. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 211-220, Padua, Italy, June 1996.
- [42] W. L. Hightower, J. F. Prins, J. H. Reif. Implementations of randomized sorting on large parallel machines. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158-167, 1992, ACM Press.
- [43] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, Vol 6, pp 249-274, 1997.

- [44] J. M. D. Hill, S. R. Donaldson, and D. Skillicorn. Portability of performance with the BSPlib communications library. *Programming Models for Massively Parallel Computers (MPPM'97)*, IEEE Computer Society Press, November 1997.
- [45] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5:10-15, 1962.
- [46] J. S. Huang and Y. C. Chow. Parallel sorting and data partitioning by sampling. *IEEE Computer Society's Seventh International Computer Software and Applications Conference*, pages 627–631, November 1983.
- [47] B. H. H. Juurlink and H. A. G. Wijshoff. The E-BSP model: Incorporating unbalanced communication and general locality into the BSP model. In *Proceedings of EURO-PAR'96*, Lyon, France, Lecture Notes in Computer Science, Springer-Verlag, August 1996.
- [48] D. E. Knuth. *The Art of Computer Programming. Volume II: Seminumerical Algorithms*. Addison-Wesley, Reading, 1969.
- [49] D. E. Knuth. *The Art of Computer Programming. Volume III: Sorting and Searching*. Addison-Wesley, Reading, 1973.
- [50] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344-354, 1985.
- [51] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes*. Morgan Kaufmann, California, 1991.
- [52] H. Li and K. C. Sevcik. Parallel Sorting by Overpartitioning. In *Proceedings of the 6-th ACM Symposium on Parallel Algorithms and Architectures*, pp. 46-56, ACM Press, 1994.
- [53] W. F. McColl. General purpose parallel computing. In *Lectures on parallel computation*, (A. Gibbons and P. Spirakis, eds.), Cambridge University Press, 1993.
- [54] W. F. McColl. Scalable parallel computing: A grand unified theory and its practical development. In *Proceedings of IFIP World Congress*, 1:539-546, Hamburg, August 1994.
- [55] W. F. McColl. An architecture independent programming model for scalable parallel computing. *Portability and Performance for Parallel Processors*, (J. Ferrante and A. J. G. Hey, eds.), John Wiley and Sons, 1994.
- [56] R. Miller. A library for Bulk-Synchronous Parallel programming. In *Proceedings of the British Computer Society Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing*, December 1993.
- [57] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the ACM*, 29(3):642-667, 1982.
- [58] C. G. Plaxton. Efficient Computation on Sparse Interconnection Networks. PhD Thesis, Department of Computer Science, Stanford University, 1989.

- [59] H. J. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34:60-76, January 1987.
- [60] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal on Computing*, 14(2):396-409, 1985.
- [61] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:362-372, 1992.
- [62] C. J. Siniolakis. On the Complexity of BSP Sorting. Technical Report PRG-TR-09-96, Computing Laboratory, Oxford University, May 1996.
- [63] D. Talia. Parallel computation still not ready for the mainstream. *Comm. of the ACM*, 44(7):98-99, July 1997.
- [64] L. G. Valiant. Bulk-synchronous parallel computers. In *Parallel Processing and Artificial Intelligence*, (M. Reeve and S. E. Zenith, eds.), Wiley, 1989.
- [65] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103-111, August 1990.
- [66] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science*, (J. van Leeuwen, ed.), North Holland, 1990.
- [67] L. G. Valiant. A combining mechanism for parallel computers. In *Parallel Architectures and Their Efficient Use*, pages 1-10, LNCS 678, Springer-Verlag, 1993.